移动端适配的常见问题总结

五、1px 问题

为了适配各种屏幕，我们写代码时一般使用设备独立像素来对页面进行布局。

而在设备像素比大于 `1` 的屏幕上，我们写的 `1px` 实际上是被多个物理像素渲染，这就会出现 `1px` 在有些屏幕上看起来很粗的现象。

5.1 border-image

基于 `media` 查询判断不同的设备像素比给定不同的 `border-image`：

```
.border_1px{
   border-bottom: 1px solid #000;
 }
 @media only screen and (-webkit-min-device-pixel-ratio:2){
     .border_1px{
         border-bottom: none;
         border-width: 0 0 1px 0;
         border-image: url(../img/1pxline.png) 0 0 2 0 stretch;
     }
 }
```

5.2 background-image

和 `border-image` 类似，准备一张符合条件的边框背景图，模拟在背景上。

```
.border_1px{
   border-bottom: 1px solid #000;
 }
 @media only screen and (-webkit-min-device-pixel-ratio:2){
     .border_1px{
         background: url(../img/1pxline.png) repeat-x left bottom;
         background-size: 100% 1px;
     }
```

```
        }
```

上面两种都需要单独准备图片，而且圆角不是很好处理，但是可以应对大部分场景。

5.3 伪类 + transform

基于 `media` 查询判断不同的设备像素比对线条进行缩放：

```
.border_1px:before{
    content: '';
    position: absolute;
    top: 0;
    height: 1px;
    width: 100%;
    background-color: #000;
    transform-origin: 50% 0%;
}
@media only screen and (-webkit-min-device-pixel-ratio:2){
    .border_1px:before{
        transform: scaleY(0.5);
    }
}
@media only screen and (-webkit-min-device-pixel-ratio:3){
    .border_1px:before{
        transform: scaleY(0.33);
    }
}
```

这种方式可以满足各种场景，如果需要满足圆角，只需要给伪类也加上 `border-radius` 即可。

5.4 svg

上面我们 `border-image` 和 `background-image` 都可以模拟 `1px` 边框，但是使用的都是位图，还需要外部引入。

借助 PostCSS 的 postcss-write-svg 我们能直接使用 border-image 和 background-image 创建 svg 的 1px 边框：

```
@svg border_1px {
  height: 2px;
  @rect {
    fill: var(--color, black);
    width: 100%;
    height: 50%;
    }
  }
.example { border: 1px solid transparent; border-image: svg(border_1px param(--color #00b1ff)) 2 2 stretch; }
```

编译后：

```
.example { border: 1px solid transparent; border-image: url("data:image/svg+xml;charset=utf-8,%3Csvg xmlns='http://www.w3.org/2000/svg' height='2px'%3E%3Crect fill='%2300b1ff' width='100%25' height='50%25'/%3E%3C/svg%3E") 2 2 stretch; }
```

上面的方案是大漠在他的文章中推荐使用的，基本可以满足所有场景，而且不需要外部引入，这是我个人比较喜欢的一种方案。

5.5 设置 viewport

通过设置缩放，让 CSS 像素等于真正的物理像素。

例如：当设备像素比为 3 时，我们将页面缩放 1/3 倍，这时 1px 等于一个真正的屏幕像素。

```
    const scale = 1 / window.devicePixelRatio;
    const viewport = document.querySelector('meta[name="viewport"]');
    if (!viewport) {
        viewport = document.createElement('meta');
        viewport.setAttribute('name', 'viewport');
        window.document.head.appendChild(viewport);
    }
    viewport.setAttribute('content', 'width=device-width,user-scalable=no,initial-scale=' + scale + ',maximum-scale=' + scale + ',minimum-scale=' + scale);
```

实际上，上面这种方案是早先 `flexible` 采用的方案。

当然，这样做是要付出代价的，这意味着你页面上所有的布局都要按照物理像素来写。这显然是不现实的，这时，我们可以借助 `flexible` 或 `vw、vh` 来帮助我们进行适配。

六、移动端适配方案

尽管我们可以使用设备独立像素来保证各个设备在不同手机上显示的效果类似，但这并不能保证它们显示完全一致，我们需要一种方案来让设计稿得到更完美的适配。

6.1 flexible 方案

`flexible` 方案是阿里早期开源的一个移动端适配解决方案，引用 `flexible` 后，我们在页面上统一使用 `rem` 来布局。

它的核心代码非常简单：

```
// set 1rem = viewWidth / 10
function setRemUnit () {
    var rem = docEl.clientWidth / 10
    docEl.style.fontSize = rem + 'px'
}
setRemUnit();
```

`rem` 是相对于 `html` 节点的 `font-size` 来做计算的。

我们通过设置 `document.documentElement.style.fontSize` 就可以统一整个页面的布局标准。

上面的代码中，将 `html` 节点的 `font-size` 设置为页面 `clientWidth`(布局视口)的 `1/10`，即 `1rem` 就等于页面布局视口的 `1/10`，这就意味着我们后面使用的 `rem` 都是按照页面比例来计算的。

这时，我们只需要将 `UI` 出的图转换为 `rem` 即可。

以 `iPhone6` 为例：布局视口为 `375px`，则 `1rem=37.5px`，这时 `UI` 给定一个元素的宽为 `75px`（设备独立像素），我们只需要将它设置为 `75/37.5=2rem`。

当然，每个布局都要计算非常繁琐，我们可以借助 `PostCSS` 的 `px2rem` 插件来帮助我们完成这个过程。

下面的代码可以保证在页面大小变化时，布局可以自适应，当触发了 `window` 的 `resize` 和 `pageShow` 事件之后自动调整 `html` 的 `fontSize` 大小。

```
1.    // reset rem unit on page resize
2.  window.addEventListener('resize', setRemUnit)window.addEventListener('pageshow', function (e) {
3.      if (e.persisted) {
4.        setRemUnit()
5.      }
6.  })
```

由于 `viewport` 单位得到众多浏览器的兼容，上面这种方案现在已经被官方弃用：

> lib-flexible 这个过渡方案已经可以放弃使用，不管是现在的版本还是以前的版本，都存有一定的问题。建议大家开始使用 viewport 来替代此方案。

下面我们来看看现在最流行的 `vh、vw` 方案。

6.2 vh、vw 方案

vh、vw 方案即将视觉视口宽度 window.innerWidth 和视觉视口高度 window.innerHeight 等分为 100 份。

上面的 flexible 方案就是模仿这种方案，因为早些时候 vw 还没有得到很好的兼容。

- vw(Viewport's width)： 1vw 等于视觉视口的 1%
- vh(Viewport's height)：1vh 为视觉视口高度的 1%
- vmin：vw 和 vh 中的较小值
- vmax：选取 vw 和 vh 中的较大值

如果视觉视口为 375px，那么 1vw=3.75px，这时 UI 给定一个元素的宽为 75px（设备独立像素），我们只需要将它设置为 75/3.75=20vw。

这里的比例关系我们也不用自己换算，我们可以使用 PostCSS 的 postcss-px-to-viewport 插件帮我们完成这个过程。写代码时，我们只需要根据 UI 给的设计图写 px 单位即可。

当然，没有一种方案是十全十美的， vw 同样有一定的缺陷：

- px 转换成 vw 不一定能完全整除，因此有一定的像素差。
- 比如当容器使用 vw， margin 采用 px 时，很容易造成整体宽度超过 100vw，从而影响布局效果。当然我们也是可以避免的，例如使用 padding 代替 margin，结合 calc()函数使用等等...

七、适配 iPhoneX

iPhoneX的出现将手机的颜值带上了一个新的高度，它取消了物理按键，改成了底部的小黑条，但是这样的改动给开发者适配移动端又增加了难度。

7.1 安全区域

在 iPhoneX发布后，许多厂商相继推出了具有边缘屏幕的手机。

这些手机和普通手机在外观上无外乎做了三个改动：圆角（ corners）、刘海（ sensor housing）和小黑条（ HomeIndicator）。为了适配这些手机，安全区域这个概念变诞生了：安全区域就是一个不受上面三个效果的可视窗口范围。

为了保证页面的显示效果，我们必须把页面限制在安全范围内，但是不影响整体效果。

7.2 viewport-fit

viewport-fit是专门为了适配 iPhoneX而诞生的一个属性，它用于限制网页如何在安全区域内进行展示。

contain：可视窗口完全包含网页内容

cover：网页内容完全覆盖可视窗口

默认情况下或者设置为 auto和 contain效果相同。

7.3 env、constant

我们需要将顶部和底部合理的摆放在安全区域内， iOS11 新增了两个 CSS 函数 env、constant，用于设定安全区域与边界的距离。

函数内部可以是四个常量：

- safe-area-inset-left：安全区域距离左边边界距离
- safe-area-inset-right：安全区域距离右边边界距离
- safe-area-inset-top：安全区域距离顶部边界距离
- safe-area-inset-bottom：安全区域距离底部边界距离

注意：我们必须指定 viweport-fit 后才能使用这两个函数：

```
<meta name="viewport" content="viewport-fit=cover">
```

constant 在 iOS<11.2 的版本中生效， env 在 iOS>=11.2 的版本中生效，这意味着我们往往要同时设置他们，将页面限制在安全区域内：

```
body {
  padding-bottom: constant(safe-area-inset-bottom);
  padding-bottom: env(safe-area-inset-bottom);
}
```

当使用底部固定导航栏时，我们要为他们设置 padding 值：

```
{
  padding-bottom: constant(safe-area-inset-bottom);
  padding-bottom: env(safe-area-inset-bottom);
}
```

八、横屏适配

很多视口我们要对横屏和竖屏显示不同的布局，所以我们需要检测在不同的场景下给定不同的样式：

8.1 JavaScript 检测横屏

`window.orientation`:获取屏幕旋转方向

```
window.addEventListener("resize", ()=>{
    if (window.orientation === 180 || window.orientation === 0) {
      // 正常方向或屏幕旋转 180 度
        console.log('竖屏');
    };
    if (window.orientation === 90 || window.orientation === -90 ){
      // 屏幕顺时钟旋转 90 度或屏幕逆时针旋转 90 度
        console.log('横屏');
    }
});
```

8.2 CSS 检测横屏

```
@media screen and (orientation: portrait) {
  /*竖屏...*/
}
@media screen and (orientation: landscape) {
  /*横屏...*/
}
```

九、图片模糊问题

9.1 产生原因

我们平时使用的图片大多数都属于位图（ `png、jpg...`），位图由一个个像素点构成的，每个像素都具有特定的位置和颜色值：

理论上，位图的每个像素对应在屏幕上使用一个物理像素来渲染，才能达到最佳的显示效果。

而在 `dpr>1` 的屏幕上，位图的一个像素可能由多个物理像素来渲染，然而这些物理像素点并不能被准确的分配上对应位图像素的颜色，只能取近似值，所以相同的图片在 `dpr>1` 的屏幕上就会模糊:

9.2 解决方案

为了保证图片质量，我们应该尽可能让一个屏幕像素来渲染一个图片像素，所以，针对不同 `DPR` 的屏幕，我们需要展示不同分辨率的图片。

如：在 `dpr=2` 的屏幕上展示两倍图 `(@2x)`，在 `dpr=3` 的屏幕上展示三倍图 `(@3x)`。

9.3 media 查询

使用 `media` 查询判断不同的设备像素比来显示不同精度的图片:

```
.avatar{
     background-image: url(conardLi_1x.png);
 }
 @media only screen and (-webkit-min-device-pixel-ratio:2){
     .avatar{
         background-image: url(conardLi_2x.png);
     }
 }
 @media only screen and (-webkit-min-device-pixel-ratio:3){
     .avatar{
         background-image: url(conardLi_3x.png);
```

```
            }
        }
```

只适用于背景图

9.4 image-set

使用 `image-set`:

```
.avatar {
    background-image: -webkit-image-set( "conardLi_1x.png" 1x, "conardLi_2x.png" 2x );
}
```

只适用于背景图

9.5 srcset

使用 `img` 标签的 `srcset` 属性，浏览器会自动根据像素密度匹配最佳显示图片:

```
<img src="conardLi_1x.png"
     srcset=" conardLi_2x.png 2x, conardLi_3x.png 3x">
```

9.6 JavaScript 拼接图片 url

使用 `window.devicePixelRatio` 获取设备像素比，遍历所有图片，替换图片地址:

```
const dpr = window.devicePixelRatio;
const images =  document.querySelectorAll('img');
images.forEach((img)=>{
  img.src.replace(".", `@${dpr}x.`);
})
```

9.7 使用 svg

SVG的全称是可缩放矢量图（ ScalableVectorGraphics）。不同于位图的基于像素， SVG 则是属于对图像的形状描述，所以它本质上是文本文件，体积较小，且不管放大多少倍都不会失真。

除了我们手动在代码中绘制 svg，我们还可以像使用位图一样使用 svg 图片：

```
<img src="conardLi.svg">

<img src="[data]">

.avatar {
  background: url(conardLi.svg);
}
```